Efficient On-the-Fly Algorithms for Partially Observable Timed Games

Franck Cassez

Efficient On-the-Fly Algorithms for Partially Observable Timed Games*

Franck Cassez

CNRS/IRCCyN
1 rue de la Noë
BP 92101
44321 Nantes Cedex 3, France
franck.cassez@cnrs.irccyn.fr

http://www.irccyn.fr/franck

Abstract. In this paper, we review some recent results on the efficient synthesis of controllers for timed systems. We first recall the basics of controller synthesis for timed games and then present an efficient on-the-fly algorithm for reachability games and its extension to partially observable timed games.

The material of this paper is based on two recent articles [13,14] that introduced truly on-the-fly algorithms for the synthesis of controllers for timed games. These results were obtained together with Alexandre David, Emmanuel Fleury and Kim G. Larsen (Aalborg University, Denmark), Didier Lime (IRCCyN, France) and Jean-François Raskin (ULB, Brussels, Belgium).

1 Introduction

The *control problem* (CP) for discrete event systems was first studied by Ramadge & Wonham in [24]. The CP is the following: "Given a finite-state model of a *plant* P (*open system)* with controllable and uncontrollable discrete actions and a *control objective* Φ, does there exist a *controller* f such that the plant supervised by f (*closed system*) satisfies Φ?"

The *dense-time* version of the CP with an *untimed* control objective has been investigated and solved in [23]. In this seminal paper, Maler *et al.* consider a plant P given by a *timed game automaton* which is a standard timed automaton [4] with its set of discrete actions partitioned into controllable and uncontrollable actions. They give an algorithm to decide whether a controller exists or not, and show that if one such controller exists, a witness can be effectively computed. In [28], Wong-Toi has given a semi-algorithm to solve the CP when the plant is defined by an extended class of timed game which is a hybrid (game) automaton.

* Work supported by the French National Research Agency ANR-06-SETI-DOTS and by the Fonds National de la Recherche Scientifique, Belgium.

The algorithms for computing controllers for timed games are based on backwards fix-point computations of the set of winning states [23,7,17]. For timed game automata, they were implemented in the tool KRONOS [2] at the end of 90's but lack efficiency because they require the computation of the complete set of winning states. Moreover the backward computation may sometimes not terminate or be very expensive for some extended classes of timed game automata, for instance if integer assignments of the form $i := j + k$ are allowed on discrete transitions.

In the last ten years, a lot of progress has been made in the design of efficient tools for the analysis (model-checking) of timed systems. Tools like KRONOS [12] or UPPAAL [21] have become very efficient and widely used to check properties of timed automata but still no real efficient counterpart had been designed for timed games.

One of the reason may be that on-the-fly algorithms have been absolutely crucial to the success of these model-checking tools. Both reachability, safety as well as general liveness properties of such timed models may be decided using on-the-fly algorithms *i.e.* by exploring the reachable state-space in a symbolic forward manner with the possibility of early termination. Timed automata technology has also been successfully applied to optimal scheduling problems with on-the-fly algorithms which quickly lead to near-optimal (time- or cost-wise) schedules [6,5,18,25,1].

Regarding timed games, in [27,3], Altisen and Tripakis have proposed a partially on-the-fly method for solving timed games. However, this method involves an extremely expensive preprocessing step in which the quotient graph of the timed game *w.r.t.* time-abstracted bisimulation[1] needs to be built. Once obtained this quotient graph may be used with any on-the-fly game-solving algorithms for untimed systems.

In a recent paper [13], we have proposed an efficient, truly on-the-fly algorithm for the computation of winning states for (reachability) timed game automata. Our algorithm is a symbolic extension of the on-the-fly algorithm suggested by Liu & Smolka in [22] for linear-time model-checking of finite-state systems. Being on-the-fly, this symbolic algorithm may terminate before having explored the entire state-space, *i.e.* as soon as a winning strategy has been identified. Also the individual steps of the algorithm are carried out efficiently by the use of so-called zones as the underlying data structure.

This algorithm has been implemented in UPPAAL-TIGA [8] which is an extension of the tool UPPAAL [21]. Some recent experiments with UPPAAL-TIGA are reported in [13] and show promising results. More recently in [14], we have extended this algorithm to deal with partially observable timed games and implemented it in a prototype based on UPPAAL-TIGA.

In this paper we focus on *reachability timed games* and present the on-the-fly algorithms of [13,14] and conclude with some current research directions.

[1] A time-abstracted bisimulation is a binary relation on states preserving discrete states and abstracted delay-transitions.

The plan of the paper is the following: in Section 2 we recall the basics of timed game automata and the backwards algorithms used to compute safety and reachability games. In Section 3 we present the efficient truly on-the-fly algorithm for reachability games that was introduced in [13] and implemented in UPPAAL-TIGA. In Section 4 we show how it can be adapted [14] to compute winning states for timed games under partial observation. Finally in Section 5 we give some current research directions.

2 Backward Algorithms for Solving Timed Games

Timed Game Automata (TGA) were introduced in [23] for solving control problems on timed systems. This section recalls the basic results for the controller synthesis for TGA. For a more complete survey the reader is referred to [10].

2.1 Notations

Let X be a finite set of real-valued variables called clocks. $\mathbb{R}_{\geq 0}$ stands for the set of non-negative reals. We note $\mathcal{C}(X)$ the set of constraints φ generated by the grammar: $\varphi ::= x \sim k \mid x-y \sim k \mid \varphi \wedge \varphi$ where $k \in \mathbb{Z}$, $x, y \in X$ and $\sim \in \{<, \leq, =, >, \geq\}$. $\mathcal{B}(X)$ is the subset of $\mathcal{C}(X)$ that uses only rectangular constraints of the form $x \sim k$. A *valuation* v of the variables in X is a mapping $v : X \to \mathbb{R}_{\geq 0}$. We let $\mathbb{R}^X_{\geq 0}$ be the set of valuations of the clocks in X. We write $\mathbf{0}$ for the valuation that assigns 0 to each clock. For $Y \subseteq X$, we denote by $v[Y]$ the valuation assigning 0 (*resp.* $v(x)$) to any $x \in Y$ (*resp.* $x \in X \setminus Y$). We denote $v + \delta$ for $\delta \in \mathbb{R}_{\geq 0}$ the valuation *s.t.* for all $x \in X$, $(v + \delta)(x) = v(x) + \delta$. For $g \in \mathcal{C}(X)$ and $v \in \mathbb{R}^X_{\geq 0}$, we write $v \models g$ if v satisfies g and $[\![g]\!]$ denotes the set of valuations $\{v \in \mathbb{R}^X_{\geq 0} \mid v \models g\}$. A *zone* Z is a subset of $\mathbb{R}^X_{\geq 0}$ *s.t.* $[\![g]\!] = Z$ for some $g \in \mathcal{C}(X)$.

2.2 Timed Automata & Simulation Graph

Definition 1 (Timed Automaton [4]). *A* Timed Automaton *(TA) is a tuple* $A = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ *where* L *is a finite set of* locations, $\ell_0 \in L$ *is the* initial *location,* Act *is the set of* actions, X *is a finite set of* real-valued clocks, $E \subseteq L \times \mathcal{B}(X) \times \mathsf{Act} \times 2^X \times L$ *is a finite set of* transitions, $\mathsf{Inv} : L \to \mathcal{B}(X)$ *associates with each location its* invariant.

A *state* of a TA is a pair $(\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ that consists of a location and a valuation of the clocks. From a state $(\ell, v) \in L \times \mathbb{R}^X_{\geq 0}$ *s.t.* $v \models \mathsf{Inv}(\ell)$, a TA can either let time progress or do a discrete transition. This is defined by the transition relation $\longrightarrow \subseteq (L \times \mathbb{R}_{\geq 0}) \times \mathsf{Act} \cup \mathbb{R}_{\geq 0} \times (L \times \mathbb{R}_{\geq 0})$ built as follows:

- for $a \in \mathsf{Act}$, $(\ell, v) \xrightarrow{a} (\ell', v')$ if there exists a transition $\ell \xrightarrow{g,a,Y} \ell'$ in E *s.t.* $v \models g$, $v' = v[Y]$ and $v' \models \mathsf{Inv}(\ell')$;
- for $\delta \geq 0$, $(\ell, v) \xrightarrow{\delta} (\ell, v')$ if $v' = v + \delta$ and $v, v' \in [\![\mathsf{Inv}(\ell)]\!]$.

Thus the semantics of a TA is the labeled transition system $S_A = (Q, q_0, \mathsf{Act} \times \mathbb{R}_{\geq 0}, \longrightarrow)$ where $Q = L \times \mathbb{R}^X_{\geq 0}$, $q_0 = (\ell_0, \mathbf{0})$ and the set of labels is $\mathsf{Act} \cup \mathbb{R}_{\geq 0}$. A *run* of a timed automaton A is a (finite or infinite) sequence of alternating time and discrete transitions in S_A. We use $\mathsf{Runs}((\ell, v), A)$ for the set of runs that start in (ℓ, v). We write $\mathsf{Runs}(A)$ for $\mathsf{Runs}((\ell_0, \mathbf{0}), A)$. If ρ is a finite run we denote $last(\rho)$ the last state of the run. An example of a timed automaton is given in Figure 1 where $[x \leq 4]$ denotes the invariant of location ℓ_4.

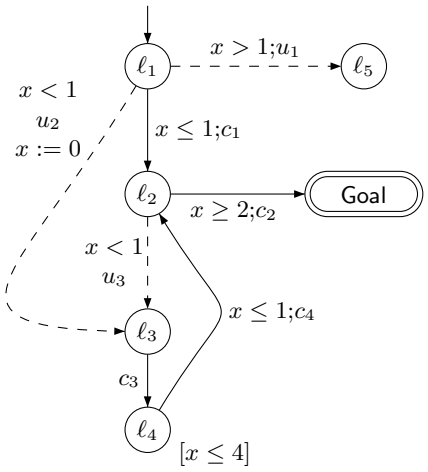

Fig. 1. A Timed Game Automaton

The analysis of TA is based on the exploration of a graph, the *simulation graph*, where the nodes are *symbolic states*. A symbolic state is a pair (ℓ, Z) where $\ell \in L$ and Z is a zone of $\mathbb{R}^X_{\geq 0}$. Let $S \subseteq Q$ and $a \in \mathsf{Act}$ we define the a-successors and a-predecessors of S respectively by:

$$\mathsf{Post}_a(S) = \{(\ell', v') \mid \exists (\ell, v) \in S, (\ell, v) \xrightarrow{a} (\ell', v')\}$$
$$\mathsf{Pred}_a(S) = \{(\ell, v) \mid \exists (\ell', v') \in S, (\ell, v) \xrightarrow{a} (\ell', v')\}.$$

The set of timed successors, $S^{\nearrow}$, of S is defined by:

$$S^{\nearrow} = \{(\ell, v + d) \mid (\ell, v) \in S \cap [\![\mathsf{Inv}(\ell)]\!], (\ell, v + d) \in [\![\mathsf{Inv}(\ell)]\!],\ d \in \mathbb{R}_{\geq 0}\}.$$

Let $\Longrightarrow$ be the relation defined on symbolic states by: $(\ell, Z) \stackrel{a}{\Longrightarrow} (\ell', Z')$ if $(\ell, g, a, Y, \ell') \in E$ and $Z' = ((Z \cap [\![g]\!])[Y])^{\nearrow}$. The simulation graph $SG(A)$ of A is given by the labeled transition system $(Z(Q), S_0, \mathsf{Act}, \Longrightarrow)$, where $Z(Q)$ is the set of zones of Q, $S_0 = (\{(\ell_0, \mathbf{0})\}^{\nearrow}) \cap [\![\mathsf{Inv}(\ell_0)]\!]$ and $\Longrightarrow$ defined as above. If A is *bounded*, *i.e.* all the clocks are bounded, the number of symbolic states is finite and $SG(A)$ is finite as well. Otherwise, a finite simulation graph that preserves for instance reachability property can be constructed for any TA. In this case, we can either transform the given TA into an equivalent one in which

all location-invariants insist on an upper bound on all clocks or, alternatively, we can apply standard extrapolation *w.r.t.* maximal constant occurring in the TA (which is correct up to time-abstracted bisimulation).

2.3 Safety and Reachability Games

Definition 2 (Timed Game Automaton [23]). *A* Timed Game Automaton (TGA) G *is a timed automaton with its set of actions* Act *partitioned into* controllable (Act_c) *and* uncontrollable (Act_u) *actions.*

The automaton in Figure 1 is also a TGA: controllable actions are depicted with plain arrows and uncontrollable ones with dashed arrows.

Given a TGA G and a set[2] of states $K \subseteq L \times \mathbb{R}^X_{\geq 0}$ the *reachability control problem* consists in finding a *strategy* f s.t. G supervised by f enforces G to enter a state in K. The *safety control problem* is the dual asking for the strategy to constantly avoid K. By "a reachability game (G, K)" (*resp.* safety) we refer to the reachability (*resp.* safety) control problem for G and K.

Let (G, K) be a reachability (resp. safety) game. Assume that all the states reachable from $(l, v) \in K$ are also in K. A finite or infinite run $\rho = (\ell_0, v_0) \xrightarrow{e_0} (\ell_1, v_1) \xrightarrow{e_1} \cdots \xrightarrow{e_n} (\ell_{n+1}, v_{n+1}) \cdots$ in $\mathsf{Runs}(G)$ is *winning* if there is some $k \geq 0$ *s.t.* $(\ell_k, v_k) \in K$ (*resp.* for all $k \geq 0$, $(\ell_k, v_k) \in K$). We rule out runs with an infinite number of consecutive time transitions of duration 0. The set of winning runs in G from (ℓ, v) is denoted $\mathsf{WinRuns}((\ell, v), G)$.

The formal definition of the control problems is based on the definitions of *strategies* and *outcomes*. A strategy [23] is a function that during the course of the game constantly gives information as to what the controller should do in order to win the game. In a given situation, the strategy could suggest the controller to either *i*) "do a particular controllable action" or *ii*) "do nothing at this point in time, just wait" which will be denoted by the special symbol λ. Let $G = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ be a TGA and $S_G = (Q, q_0, \rightarrow)$ its semantics.

Definition 3 (Strategy). *A* strategy f *over* G *is a partial function from the finite runs of* $\mathsf{Runs}(G)$ *to* $\mathsf{Act}_c \cup \{\lambda\}$ s.t. *for every finite run* ρ

- *if* $f(\rho) \in \mathsf{Act}_c$ *then* $last(\rho) \xrightarrow{f(\rho)} (\ell', v')$ *for some* (ℓ', v') *and*
- *if* $f(\rho) = \lambda$ *then* $last(\rho) \xrightarrow{\delta} (\ell', v')$ *for some* $\delta > 0$ *and* (ℓ', v').

We denote $\mathsf{Strat}(G)$ the set of strategies over G. A strategy f is *state-based* if $\forall \rho, \rho' \in \mathsf{Runs}(G), last(\rho) = last(\rho')$ implies that $f(\rho) = f(\rho')$. State-based strategies are also called *memoryless* strategies in game theory [17,26].

The restricted behavior of a TGA G controlled with some strategy f is defined by the notion of *outcome* [17].

[2] For real computation we shall require that K is defined as a finite union of symbolic states.

Definition 4 (Outcome). *Let f be a strategy over G. The (set of)* outcomes $\mathsf{Outcome}(q, f)$ *of f from q in S_G is the subset of* $\mathsf{Runs}(q, G)$ *defined inductively by:*

- $q \in \mathsf{Outcome}(q, f)$,
- *if* $\rho \in \mathsf{Outcome}(q, f)$ *then* $\rho' = \rho \xrightarrow{e} q' \in \mathsf{Outcome}(q, f)$ *if* $\rho' \in \mathsf{Runs}(q, G)$ *and one of the following three conditions hold:*
  1. $e \in \mathsf{Act}_u$,
  2. $e \in \mathsf{Act}_c$ *and* $e = f(\rho)$,
  3. $e \in \mathbb{R}_{\geq 0}$ *and* $\forall 0 \leq e' < e, \exists q'' \in Q$ s.t. $last(\rho) \xrightarrow{e'} q'' \wedge f(\rho \xrightarrow{e'} q'') = \lambda$.
- *for an infinite run ρ, $\rho \in \mathsf{Outcome}(q, f)$ if all the finite prefixes of ρ are in* $\mathsf{Outcome}(q, f)$.

We assume that uncontrollable actions can only spoil the game and the controller has to do some controllable action to win [7,23,18]. In other words, an uncontrollable action cannot be forced to happen in G. Thus, a run may end in a state where only uncontrollable actions can be taken. For reachability games we assume w.l.o.g. that the goal is a particular location Goal *i.e.* $K = \{(\mathsf{Goal}, v) \,|\, v \in \mathbb{R}^X_{\geq 0}\}$ as depicted on Figure 1. For safety games we have to avoid a particular location Bad *i.e.* $K = \{(\mathsf{Bad}, v) \,|\, v \in \mathbb{R}^X_{\geq 0}\}$.

In the sequel we focus on *reachability games*. A *maximal run* ρ is either an infinite run (supposing no infinite sequence of delay transitions of duration 0) or a finite run ρ that satisfies either *i)* $last(\rho) \in K$ or *ii)* if $\rho \xrightarrow{a}$ then $a \in \mathsf{Act}_u$ *i.e.* the only possible discrete actions from $last(\rho)$ (if any) are uncontrollable actions. A strategy f is *winning* from q if all maximal runs in $\mathsf{Outcome}(q, f)$ are in $\mathsf{WinRuns}(q, G)$. A state q in a TGA G is *winning* if there exists a winning strategy f from q in G. We denote by $\mathcal{W}(G)$ the set of winning states in G and $\mathsf{WinStrat}(q, G)$ the set of winning strategies from q over G.

2.4 Backward Algorithms for Solving Timed Games

Let $G = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ be a TGA. For timed games, the computation of the winning set of states is based on the definition of a *controllable predecessors* operator [17,23]. The controllable and uncontrollable discrete predecessors of $S \subseteq Q$ are defined by $\mathsf{cPred}(S) = \bigcup_{c \in \mathsf{Act}_c} \mathsf{Pred}_c(S)$ and $\mathsf{uPred}(S) = \bigcup_{u \in \mathsf{Act}_u} \mathsf{Pred}_u(S)$. A notion of *safe* timed predecessors of a set S *w.r.t.* a set U is also needed. Intuitively a state q is in $\mathsf{Pred}_t(S, U)$ if from q we can reach $q' \in S$ by time elapsing and along the path from q to q' we avoid U. This operator is formally defined by:

$$\mathsf{Pred}_t(S, U) = \{q \in Q \mid \exists \delta \in \mathbb{R}_{\geq 0} \text{ s.t. } q \xrightarrow{\delta} q', q' \in S \text{ and } \mathsf{Post}_{[0,\delta]}(q) \subseteq \overline{U}\} \quad (1)$$

where $\mathsf{Post}_{[0,\delta]}(q) = \{q' \in Q \mid \exists t \in [0, \delta] \text{ s.t. } q \xrightarrow{t} q'\}$ and $\overline{U} = Q \setminus U$. The *controllable predecessors* operator π is formally defined as follows (Figure 2):

$$\pi(S) = \mathsf{Pred}_t\left(S \cup \mathsf{cPred}(S), \mathsf{uPred}(\overline{S})\right) \quad (2)$$

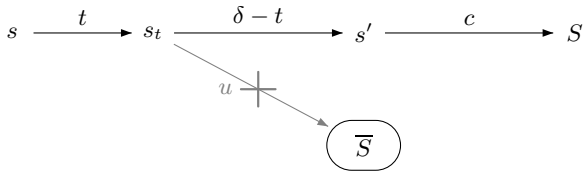

Fig. 2. $\pi(S)$

Note that according to this definition, even forced uncontrollable actions (*e.g.* by an invariant) are not bound to happen and cannot help to win. A controllable action must be taken to reach a winning state and uncontrollable actions can only spoil the game.

If S is a finite union of symbolic states, then $\pi(S)$ is again a finite union of symbolic states and $\pi(S)$ is effectively computable. Assume (G, K) is a reachability game. In this case the least fix-point of $S = K \cup \pi(S)$ can be computed by the iterative process given by $W^0 = K$ and $W^{n+1} = W^n \cup \pi(W^n)$. This computation will converge after finitely many steps for TGA [23] and the we denote W^* the fix-point. As it is proved in [23], $W^* = \mathcal{W}(G)$. Note that W^* is the maximal (complete) set of winning states of G *i.e.* a state is winning iff it is in W^*. Thus there is a winning strategy in G iff $(\ell_0, \mathbf{0}) \in W^*$. Altogether this gives a symbolic algorithm for solving reachability games. For safety games, it suffices to take the greatest fix-point W^* of $S = K \cap \pi(S)$ and again $W^* = \mathcal{W}(G)$.

Another important result for reachability and safety TGA is that memoryless strategies are sufficient to win [7,23]. This makes it possible to compute a most permissive state-based strategy. Extracting strategies can be done using the set of winning states W^* (see [9] for reachability games).

For the example of Figure 1, the set of symbolic winning states is given by: $\mathcal{W} = \{(\ell_1, x \leq 1), (\ell_2, x \leq 2), (\ell_3, x \leq 1), (\ell_4, x \leq 1), (\mathsf{Goal}, x \geq 0)\}$.

A winning strategy would consist in taking c_1 immediately in all states (ℓ_1, x) with $x \leq 1$; taking c_2 immediately in all states (ℓ_2, x) with $x \leq 2$; taking c_3 immediately in all states (ℓ_3, x) and delaying in all states (ℓ_4, x) with $x < 1$ until the value of x is 1 at which point the edge c_4 is taken.

3 On-the-Fly Algorithm for Reachability Games

For finite-state systems, on-the-fly model-checking algorithms has been an active and successful research area since the end of the 80's, with the algorithm proposed by Liu & Smolka [22] being particularly elegant (and optimal).

In [13], we have proposed a version of this algorithm for timed games. In the sequel we first present the on-the-fly algorithm for reachability *untimed* games and then show how to design a symbolic version for *timed* games.

3.1 On-the-Fly Algorithm for Discrete Games

Untimed games are a restricted class of timed games with only finitely many states Q and with only discrete actions, *i.e.* the set of labels in the semantics of

the game is Act. Hence (memoryless) strategies amounts to choosing a controllable action given the current state, *i.e.* $f : Q \to \mathsf{Act}_c$. For (untimed) reachability games we assume a designated location Goal and the purpose of the analysis is to decide the existence of a strategy f where all runs contains Goal.

The on-the-fly algorithm, OTFUR, we have proposed in [13] is given in Fig. 3. The idea for this algorithm is the following: we assume that some variables store two sets of transitions: *ToExplore* store the transitions that have explored and *ToBackPropagate* store the transitions the target states of which has been declared winning. Another variable, $Passed$, stores the set of states that have already been encountered. Each encountered state $q \in Passed$ has a status, $Win[q]$ which is either *winning* (1) or *unknown* (0). We also use a variable $Depend[q]$ that stores for each q, the set of explored transitions t s.t. q is a target of t. The initial values of the variables are set by lines 2 to 6.

To perform a step of the on-the-fly algorithm OTFUR, we pick transition a (q, α, q') in $ToExplore \cup ToBackPropagate$ (line 10) and process it as follows:

- if the target state q' is encountered for the first time ($q' \notin Passed$), we update $Passed$, $Depend$ and $Win[q']$ (lines 14–16). We also initialize some counters (lines 12 and 13) $c(q')$ and $u(q')$ which have the following meaning: at each time, $c(q')$ represents the number of controllable transitions that can be taken to reach a winning state from q' and $u(s)$ represents the number of uncontrollable hazardous transitions from q' *i.e.* those for which we do not know yet if they lead to a winning state. When q' is first encountered $u(q')$ is simply the number of outgoing uncontrollable transitions from q'. Finally (lines 17 to 20), depending on the status of q' we add the outgoing transitions to *ToExplore* or just schedule the current transition for back propagation if q' is winning.
- in case $q' \in Passed$, it means that either its status has been changed recently (and we just popped a transition from *ToBackPropagate*) or that a new transition leading to q' has been chosen (from *ToExplore*). We thus check whether the status of q' is winning and if yes, we update some information on q: lines 24 and 25 updates the counters $c(q)$ or $u(q)$ depending on the type of the transition being processed (controllable or not). The state q can be declared winning (line 27) if at least one controllable transition leads to a winning state ($c(q) \geq 1$) and all outgoing uncontrollable transitions lead to a winning state as well ($u(q) = 0$). In this case the transitions leading to q ($Depend[q]$) are scheduled for back propagation (line 29). Otherwise we have just picked a new transition leading to q' and we only update $Depend[q']$ (line 31).

The correctness proof of this algorithm is given by the following theorem:

Theorem 1 ([13]). *Upon termination of* OTFUR *on a given untimed game G the following holds:*

1. *If $q \in Passed$ and $Win[q] = 1$ then $q \in \mathcal{W}(G)$;*
2. *If $(ToExplore \cup ToBackPropagate) = \emptyset$ and $Win[q] = 0$ then $q \notin \mathcal{W}(G)$.*

```
1:  Initialization
2:    Passed ← {q0};
3:    ToExplore ← {(q0, α, q') | α ∈ Act, q -α-> q'};
4:    ToBackPropagate ← ∅;
5:    Win[q0] ← (q0 = Goal ? 1 : 0); // set status to 1 if q0 is Goal
6:    Depend[q0] ← ∅;
7:  Main
8:    while ((ToExplore ∪ ToBackPropagate ≠ ∅)) ∧ Win[q0] ≠ 1)) do
9:        // pick a transition from ToExplore or ToBackPropagate
10:       e = (q, α, q') ← pop(ToExplore) or pop(ToBackPropagate);
11:       if q' ∉ Passed then
12:           c(q') = 0;
13:           u(q') = |{(q' -a-> q'', a ∈ Act_u}|;
14:           Passed ← Passed ∪ {q'};
15:           Depend[q'] ← {(q, α, q')};
16:           Win[q'] ← (q' = Goal ? 1 : 0);
17:           if Win[q'] = 0 then
18:               ToExplore ← ToExplore ∪ {(q', α, q'') | q' -α-> q''};
19:           else
20:               ToBackPropagate ← ToBackPropagate ∪ {e};
21:       else
22:           if Win[q'] = 1 then
23:              // update the counters of the state q
24:              if α ∈ Act_c then c(q) ← c(q) + 1;
25:              else u(q) ← u(q) − 1;
26:              // re-evaluate the status of the state q
27:              Win[q] ← (c(q) ≥ 1) ∧ (u(q) = 0);
28:              if Win[q] then
29:                  ToBackPropagate ← ToBackPropagate ∪ Depend[q];
30:           else // Win[q'] = 0
31:              Depend[q'] ← Depend[q'] ∪ {e};
32:           endif
33:       endif
34:   endwhile
```

Fig. 3. OTFUR: **O**n-**T**he-**F**ly Algorithm for **U**ntimed **R**eachability Games

In addition to being on-the-fly and correct, this algorithm terminates and is optimal in that it has linear time complexity in the size of the underlying untimed game: it is easy to see that each edge $e = (q, \alpha, q')$ will be added to *ToExplore* at most once and to *ToBackPropagate* at most once as well, the first time q is encountered (and added to $Passed$) and the second time when $Win[q']$ changes winning status from 0 to 1. Notice that to obtain an algorithm running in linear time in the size of G (*i.e.* $|Q| + |E|$) it is important that the reevaluation of the winning status of a state q is performed using the two variables $c(q)$ and $u(q)$.

3.2 On-the-Fly Algorithm for Timed Games

We can extend algorithm OTFUR to the timed case using a zone-based forward and on-the-fly algorithm for solving timed reachability games. The algorithm, SOTFTR, is given in Fig. 4 and may be viewed as an interleaved combination of *forward computation* of the *simulation graph* of the timed game automaton together with *back-propagation* of information *of winning states*. As in the untimed case the algorithm is based on two sets, *ToExplore* and *ToBackPropagate*, of symbolic edges in the simulation-graph, and a passed-list, $Passed$, containing all the symbolic states of the simulation-graph encountered so far by the algorithm. The crucial point of our symbolic extension is that the winning status of an individual symbolic state is no more 0 or 1 but is now the *subset* $Win[S] \subseteq S$ (union of zones) of the symbolic state S which is currently known to be winning. The set $Depend[S]$ indicates the set of edges (or predecessors of S) which must be reevaluated (*i.e.* added to *ToBackPropagate*) when new information about $Win[S]$ is obtained, *i.e.* when $Win[S] \subsetneq Win^*$. Whenever a symbolic edge $e = (S, \alpha, S')$ is considered with $S' \in Passed$, the edge e is added to the dependency set of S' so that that possible future information about additional winning states within S' may also be back-propagated to S. In Table 1, we illustrate the forward exploration and backwards propagation steps of the algorithm.

The correctness of the symbolic on-the-fly algorithm SOTFTR is given by the theorem:

Theorem 2 ([13]). *Upon termination of the algorithm* SOTFTR *on a given timed game automaton G the following holds:*

1. *If $(\ell, v) \in Win[S]$ for some $S \in Passed$ then $(\ell, v) \in \mathcal{W}(G)$;*
2. *If ToExplore $\cup$ ToBackPropagate $= \emptyset$, $(\ell, v) \in S \setminus Win[S]$ for some $S \in Passed$ then $(\ell, v) \notin \mathcal{W}(G)$.*

Termination of the algorithm SOTFTR is guaranteed by the finiteness of the number of symbolic states of $SG(A)$. Moreover, each edge (S, α, T) will be present in the *ToExplore* and *ToBackPropagate* at most $1 + |T|$ times, where $|T|$ is the number of regions of T: (S, α, T) will be in *ToExplore* the first time that S is encountered and subsequently in *ToBackPropagate* each time the set $Win[T]$ increases. Now, any given region may be contained in several symbolic states of the simulation graph (due to overlap). Thus the SOTFTR algorithm

1: **Initialization**
2: $Passed \leftarrow \{S_0\}$ where $S_0 = \{(\ell_0, \mathbf{0})\}^{\nearrow}$;
3: $ToExplore \leftarrow \{(S_0, \alpha, S') \mid S' = \mathsf{Post}_\alpha(S_0)^{\nearrow}\}$;
4: $ToBackPropagate \leftarrow \emptyset$;
5: $Win[S_0] \leftarrow S_0 \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
6: $Depend[S_0] \leftarrow \emptyset$;
7: **Main**
8: **while** $((ToExplore \cup ToBackPropagate \neq \emptyset) \wedge (\ell_0, \mathbf{0}) \notin Win[S_0]))$ **do**
9: *// pick a transition from ToExplore or ToBackPropagate*
10: $e = (S, \alpha, S') \leftarrow pop(ToExplore)$ or $pop(ToBackPropagate)$;
11: **if** $S' \notin Passed$ **then**
12: $Passed \leftarrow Passed \cup \{S'\}$;
13: $Depend[S'] \leftarrow \{(S, \alpha, S')\}$;
14: $Win[S'] \leftarrow S' \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
15: **if** $Win[S'] \subsetneq S'$ **then**
16: $ToExplore \leftarrow ToExplore \cup \{(S', \alpha, S'') \mid S'' = \mathsf{Post}_\alpha(S')^{\nearrow}\}$;
17: **if** $Win[S'] \neq \emptyset$ **then**
18: $ToBackPropagate \leftarrow ToBackPropagate \cup \{e\}$;
19: **else**
20: *// If $T \notin Passed$, we assume $Win[T] = \emptyset$*
21: $Good \leftarrow Win[S] \cup \bigcup_{S \xrightarrow{c} T} \mathsf{Pred}_c(Win[T])$;
22: $Bad \leftarrow \bigcup_{S \xrightarrow{u} T} \mathsf{Pred}_u(T \setminus Win[T])) \cap S$;
23: $Win^* \leftarrow \mathsf{Pred}_t(Good, Bad)$;
24: **if** $(Win[S] \subsetneq Win^*)$ **then**
25: $Waiting \leftarrow Waiting \cup Depend[S]$;
26: $Win[S] \leftarrow Win^*$;
27: $Depend[S'] \leftarrow Depend[S'] \cup \{e\}$;
28: **endif**
29: **endwhile**

Fig. 4. SOTFTR: **S**ymbolic **O**n-**T**he-**F**ly Algo. for **T**imed **R**eachability Games

Steps			*ToExplore* ∪ *ToBackPropagate*	*Passed*	*Depend*	*Win*
#	S	S'				
0	-	-	$(S_0, u_1, S_1), (S_0, u_2, S_2), (\mathbf{S_0}, \mathbf{c_1}, \mathbf{S_3})$	S_0	-	$(S_0, \emptyset)$
1	S_0	S_3	$(S_0, u_1, S_1), (S_0, u_2, S_2)$ + $(\mathbf{S_3}, \mathbf{c_1}, \mathbf{S_4}), (S_3, u_3, S_2)$	S_3	$S_3 \mapsto (S_0, c_1, S_3)$	$(S_3, \emptyset)$
2	S_3	S_4	$(S_0, u_1, S_1), (S_0, u_2, S_2), (S_3, u_3, S_2)$ + $(\mathbf{S_3}, \mathbf{c_2}, \mathbf{S_4})$	S_4	$S_4 \mapsto (S_3, c_2, S_4)$	(S_4, S_4)
3	S_3	S_4	$(S_0, u_1, S_1), (S_0, u_2, S_2), (S_3, u_3, S_2)$ + $(\mathbf{S_0}, \mathbf{c_1}, \mathbf{S_3})$	-	-	$(S_3, x \geq 1)$
4	S_0	S_3	$(S_0, u_1, S_1), (S_0, u_2, S_2), (\mathbf{S_3}, \mathbf{u_3}, \mathbf{S_2})$	S_4	$S_3 \mapsto (S_0, c_1, S_3)$	$(S_0, x = 1)$
5	S_3	S_2	$(S_0, u_1, S_1), (S_0, u_2, S_2)$ + $(\mathbf{S_2}, \mathbf{c_3}, \mathbf{S_5})$	S_2	$S_2 \mapsto (S_3, u_3, S_2)$	$(S_2, \emptyset)$
6	S_2	S_5	$(S_0, u_1, S_1), (S_0, u_2, S_2)$ + $(\mathbf{S_5}, \mathbf{c_4}, \mathbf{S_3})$	S_5	$S_5 \mapsto (S_2, c_3, S_2)$	$(S_5, \emptyset)$
7	S_5	S_3	$(S_0, u_1, S_1), (S_0, u_2, S_2)$ + $(\mathbf{S_2}, \mathbf{c_3}, \mathbf{S_5})$	-	$S_3 \mapsto \begin{matrix}(S_2, c_3, S_2)\\(S_5, c_4, S_3)\end{matrix}$	$(S_5, x \leq 1)$
8	S_2	S_5	$(S_0, u_1, S_1), (S_0, u_2, S_2)$ + $(\mathbf{S_3}, \mathbf{u_3}, \mathbf{S_2})$	-	$S_5 \mapsto (S_2, c_3, S_2)$	$(S_2, x \leq 1)$
9	S_3	S_2	$(S_0, u_1, S_1), (\mathbf{S_0}, \mathbf{u_2}, \mathbf{S_2})$ + $(S_0, c_1, S_3), (S_5, c_4, S_3)$	-	-	(S_3, S_3)
10	S_0	S_2	$(S_0, u_1, S_1), (S_0, c_1, S_3), (\mathbf{S_5}, \mathbf{c_4}, \mathbf{S_3})$	-	$S_2 \mapsto \begin{matrix}(S_3, u_3, S_2)\\(S_0, u_2, S_2)\end{matrix}$	$(S_0, x \leq 1)$
11	S_5	S_3	$(S_0, u_1, S_1), (\mathbf{S_0}, \mathbf{c_1}, \mathbf{S_3})$	-	-	-
12	S_0	S_3	$(\mathbf{S_0}, \mathbf{u_1}, \mathbf{S_1})$	-	-	-
13	S_0	S_1	$\emptyset$	S_1	$S_1 \mapsto (S_0, u_1, S_1)$	$(S_1, \emptyset)$

At step n, $(\mathbf{S}, \alpha, \mathbf{S'})$ is the transition popped at step $n+1$;
At step n, $+(S, \alpha, S')$ the transition added to *ToBackPropagate* or *ToExplore* at step n;
Symbolic States: $S_0 = (\ell_1, x \geq 0)$,$S_1 = (\ell_5, x > 1)$, $S_2 = (\ell_3, x \geq 0)$, $S_3 = (\ell_2, x \geq 0)$,
$S_4 = (\mathsf{Goal}, x \geq 2)$, $S_5 = (\ell_4, x \geq 0)$

Table 1. Running SOTFTG

is *not* linear in the region-graph and hence not theoretically optimal, as an algorithm with linear worst-case time-complexity could be obtained by applying the untimed algorithm directly to the region-graph. However, this is only a theoretical result and it turns out that the implementation of the algorithm in UPPAAL-TIGA is very efficient.

We can optimize (space-wise) the previous algorithm. When we explore the automaton forward, we check if any newly generated symbolic state S' belongs *Passed*. As an optimization we may instead use the classical inclusion check: $\exists S'' \in \mathit{Passed}$ s.t. $S' \subseteq S''$, in which case, S' is discarded and we update $\mathit{Depend}[S'']$ instead. Indeed, new information learned for S'' can be new information on S' but not necessarily. This introduces an overhead (time-wise) in the sense that we may back-propagate irrelevant information. On the other hand, back-propagating only the relevant information would be unnecessarily complex and would void most of the memory gain introduced by the use of inclusion. In practice, the reduction of the number of forward steps obtained by the inclusion check pays off for large systems and is a little overhead otherwise, as shown in our experiments. It is also possible to propagate information about losing states: in the case of reachability games, if a state is a deadlock state and is not winning, for sure it is losing. This can also speed-up the algorithm. For the example of Figure 1, we can propagate the information that $(\ell_5, x > 1)$ is losing which

entails $(\ell_1, x > 1)$ is losing as well. Then it only remains to obtain the status of $(\ell_1, x \leq 1)$ to determine if the game is winning or not.

4 On-the-Fly Algorithm for Partially Observable Games

4.1 Partial Observability

In the previous sections we have assumed that the controller has perfect information about the system: at any time, the controller will know precisely in what state the system is. In general however — e.g. due to limited sensors — a controller will only have imperfect (or partial) information about the state of the environment. For instance some uncontrollable actions may be *unobservable*. In the discrete case it is well known how to handle partial observability of actions as it roughly amounts to determinize a finite state system.

However for the timed case under partial observability of events, it has been shown in [11] that the controller synthesis problem is in general undecidable. Fixing the resources of the controller (i.e. a maximum number of clocks and maximum allowed constants in guards) regains decidability [11], a result which also follows from the quotient and model construction results of [19,20].

Another line of work [16,29] recently revisited partial observability for finite state systems. This time, the partial observability amounts to imperfect information on the state of the system. Only a finite number of possible *observations* can be made on the system configurations and this provides the sole basis for the strategy of the controller. The framework of [16,29] is essentially turn-based. Moreover, the controller can make an observation after each discrete transition. It could be that it makes the same observation several times in a row, being able to count the number of steps that have been taken by the system.

If we want to extend this work to timed systems, we need to add some more constraints. In particular, the strategy of the controller will have to be *stuttering invariant*, i.e. the strategy cannot be affected by a sequence of environment or time steps unless changes in the observations occur. In this sense the strategy is "triggered" by the changes of observations.

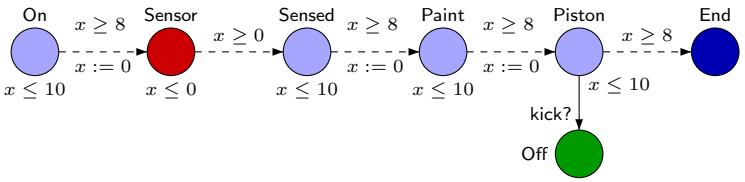

Fig. 5. Timed Game with Imperfect Information.

To illustrate the concepts of imperfect information and stuttering invariance consider the timed game automaton in Figure 5 modelling a production system

for *painting* a box moving on a conveyor belt. The various locations indicate the position of the box in the system: in Sensor a sensor is assumed to reveal the presence of the box, in Sensed the box is moving along the belt towards the painting area, in Paint the actual painting of the box takes place, in Piston the box may be kick?'ed off the belt leading to Off; if the box is not kicked off it ends in End. All phases are assumed to last between 8 and 10 seconds, except for the phase Sensor, which is instantaneous. The uncontrollability of this timing uncertainty is indicated by the dashed transitions between phases. The controller should now issue a *single* kick?'command at the appropriate moment in order to guarantee that the box will — regardless of the above timing uncertainty — be kicked off the belt. However the controller has *imperfect information* of the position of the box in the system. In particular, the controller cannot directly observe whether the box is in the Sensed, Paint or in the Piston phase nor can the value of the clock x be observed. Still equipping the controller with its own clock y – which it may reset and test (against a finite number of predicates) – it might be possible to synthesize a control strategy despite having only partial information: in fact it may be deduced that the box will definitely be in the Piston area within 20-24 seconds after being sensed. In contrast, an increased timing uncertainty where a phase may last between 6 and 10 seconds will make a single-kick? strategy impossible.

4.2 Observation-Based Stuttering Invariant Strategies

In the untimed setting of [16,29], each state change produces a new observation. For instance, if a run of the system is

$$\rho = l_0 \xrightarrow{a_0} l_1 \xrightarrow{a_1} l_2 \cdots \xrightarrow{a_n} l_{n+1}$$

we make the observation $\mathsf{Obs}(\rho) = \mathsf{Obs}(l_0)\mathsf{Obs}(l_1)\cdots\mathsf{Obs}(l_{n+1})$ where Obs is a mapping from states to a finite set of observations. In the previous example, even if Sensed, Paint and Piston produce the same observation, say 01, we could deduce where the box is by counting the number of 01.

Also, if each state change produces a new observation in the untimed setting, it cannot be a realistic assumption in the timed setting: time is continuous and the state of the system continuously changes.

A more realistic assumption about the system under observation is that the controller can only see *changes* of observations. In the previous piston example, assume the system makes the following steps:

$$(\mathsf{On}, x=0) \xrightarrow{8} (\mathsf{On}, x=8) \rightarrow (\mathsf{Sensor}, x=0) \rightarrow (\mathsf{Sensed}, x=0)\cdots \\ \cdots \xrightarrow{9} (\mathsf{Sensed}, x=9) \rightarrow (\mathsf{Paint}, x=0)$$

The sequence of observations the controller makes is: On Sensor 01 if $\mathsf{Obs}(\mathsf{On}) = \mathsf{On}$, $\mathsf{Obs}(\mathsf{Sensor}) = \mathsf{Sensor}$ and $\mathsf{Obs}(\mathsf{Sensed}) = \mathsf{Obs}(\mathsf{Paint}) = 01$.

This is why in [14] we consider *stuttering-free* observations. We assume we are given a finite set of observations $\mathcal{O} = \{o_1, o_2, \cdots, o_k\}$ and a mapping Obs :

$L \times \mathbb{R}^X_{\geq 0} \to \mathcal{O}$ (*i.e.* from the state space of the timed game automaton to $\mathcal{O}$). Given a run of timed game automaton

$$\rho = (l_0, v_0) \xrightarrow{e_0} (l_1, v_1) \xrightarrow{e_1} (l_2, v_2) \cdots \xrightarrow{e_n} (l_{n+1}, v_{n+1}) \cdots$$

we can define the observation of ρ by the sequence:

$$\mathsf{Obs}(\rho) = \mathsf{Obs}(l_0, v_0)\mathsf{Obs}(l_1, v_1) \cdots \mathsf{Obs}(l_{n+1}, v_{n+1}) \cdots$$

The stuttering-free observation of ρ is $\mathsf{Obs}^*(\rho)$ and is obtained from $\mathsf{Obs}(\rho)$ by collapsing successive identical observations $o_1 o_1 \cdots o_1$ into one o_1. In this setting the controller has to make a decision on what to do after a finite run ρ, according to the stuttering-free observation of ρ. Let f be a strategy in $\mathsf{Strat}(G)$. f is (observation based) *stuttering invariant* if for all ρ, ρ', finite runs in $\mathsf{Runs}(G)$, if $\mathsf{Obs}^*(\rho) = \mathsf{Obs}^*(\rho')$ then $f(\rho) = f(\rho')$.

The control problem under partial observation thus becomes: given a reachability game (G, K), is there an observation based stuttering invariant strategy to win (G, K)?

This raises an issue about the shapes of observations in timed systems: assume for the piston game, we define two observations $o_1 = (\mathsf{On}, x \leq 3)$ and $o_2 = (\mathsf{On}, x > 3)$. Given any finite run $(\mathsf{On}, x = 0) \xrightarrow{r} (\mathsf{On}, x = r)$ with $r \leq 10$ there is one stuttering-free observation: either o_1 if $r \leq 3$ or $o_1 o_2$ if $r > 3$. Still we would like our controller to be able to determine its strategy right after each change of observations, *i.e.* the controller continuously monitors the observations and can detect rising edges of each new observation. This implies that a *first instant* exists where a change of observations occurs. To ensure this we can impose syntactic constraints on the shape of the zones that define observations. They must be conjunctions of constraints of the form $k_1 \leq x < k_2$ where x is a clock and $k_1, k_2 \in \mathbb{N}$.

4.3 Playing with Stuttering Invariant Strategies

The controller has to play according to *(observation based) stuttering invariant strategies* (OBSI strategies for short). Initially and whenever the current observation of the system state changes, the controller either proposes a controllable action $c \in \mathsf{Act}_c$, or the special action λ (do nothing *i.e.* delay). When the controller proposes $c \in \mathsf{Act}_c$, this intuitively means that he wants to play the action c as soon as this action is enabled in the system. When he proposes λ, this means that he does not want to play any discrete actions until the next change of observation, he is simply waiting for the next observation. Thus, in the two cases, the controller sticks to his choice until the observation of the system changes: in this sense he is playing with an observation based stuttering invariant strategy. Once the controller has committed to a choice, the environment decides of the evolution of the system until the next observation. The game can be thought of to be a two-player game with the following rules:

1. if the choice of the controller is a discrete action $c \in \mathsf{Act}_c$, the environment can choose to play, as long as the observation does not change, either (i) discrete actions in $u \in \mathsf{Act}_u \cup \{c\}$ or (ii) let time elapse as long as c is not enabled. Thus it can produce sequences of discrete and time steps that respect the (i) and (ii) with action c being urgent,
2. if the choice of the controller is the special action λ the environment can choose to play, as long as the observation does not change, any of its discrete actions in Act_u or let time pass; it can produce sequences of discrete and time steps respecting the previous constraints;
3. the turn is back to the controller as soon as the next observation is reached. We have imposes special shape of constraints to ensure that a next first new observation always exists.

The previous scheme formalizes the intuition of observation based stuttering invariant strategies.

To solve the control problem for G under partial observation given by a finite set $\mathcal{O}$, we reduce it to a control problem on a new game G' under full observation.

4.4 An Efficient Algorithm for Partially Observable Timed Games

The reduction we have proposed in [14] follows the idea of *knowledge based subset construction* for discrete games introduced in [16,29].

Let $G = (L, \ell_0, \mathsf{Act}, X, E, \mathsf{Inv})$ be a TGA, and $\mathsf{Obs} : L \times \mathbb{R}^X_{\geq 0} \to \mathcal{O}$ be an observation map. Let K be particular location such that (G, K) is a reachability game and there is an observation o s.t. $\mathsf{Obs}(s) = o \iff s \in K$ *i.e.* the controller can observe if the system is in a winning state. We use $\mathsf{Obs}(K)$ for this particular observation. From G we build a finite discrete game $\tilde{G}$ the states of which are unions of pairs of symbolic states (l, Z) ($\ell \in L$ and Z is a zone). Moreover, we require that each state of $\tilde{G}$ contains pairs (l, Z) that have the same observation.

Each set of states S of $\tilde{G}$ corresponds to a set of points where, in the course of the game G, the controller can choose a new action because a new observation has just been seen. The controller can choose either to do a $c \in \mathsf{Act}_c$ or to let time pass (λ). Once the controller has made a choice, it cannot do anything until a new observation occurs. Given a state $(l, v) \in S$ and a choice a of the controller, we can define the tree $Tree((l, v), a)$ of possible runs starting in (l, v): this tree is just the unfolding of the game where we keep only the branches with actions in $\mathsf{Act}_u \cup \{a\}$ (see Figure 6, left). In this section, we assume that every finite run can be extended in an infinite run. Then on each infinite branch of this tree,

1. either there is a state with an observation o' different from $\mathsf{Obs}(l, v)$. In this case we can define the first (time-wise) state with a new observation. Such an example is depicted on Figure 6 (left): from (l, v), there is a first state with a new observation in o_1 and in o_2. Because we require that our observations have special shapes, this first state always exists.
2. or all the states have the same observation on the branch: this is depicted by the infinite path starting from (l', v').

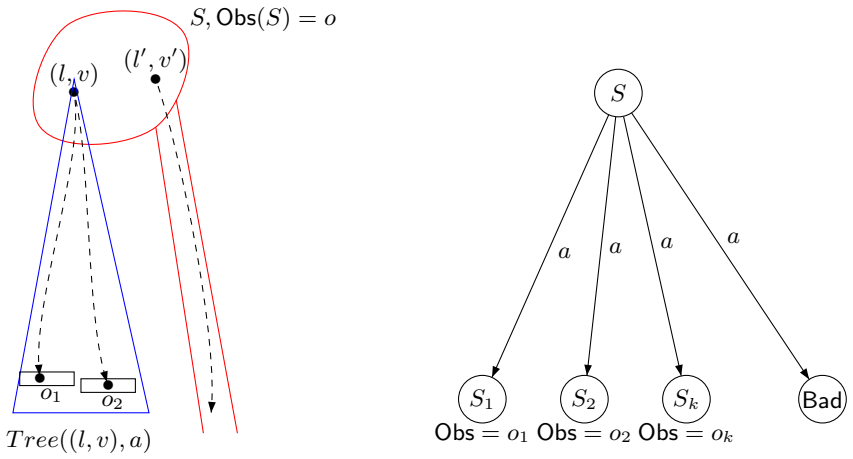

Fig. 6. From G to $\tilde{G}$

We denote $\mathsf{Next}_a(l, v)$ the set of first (time-wise) states with a new observation that can be reached from (l, v) if the controller plays a. If there is an infinite run from (l', v') on which all the states have the same observation, we say that (l', v') is a *sink* state for a.

We can now define the game $\tilde{G}$ as follows (Figure 6, right):

- the initial state of $\tilde{G}$ is $\{(\ell_0, \mathbf{0})\}$;
- let S be a state of $\tilde{G}$ with an observation different from $\mathsf{Obs}(K)$ (not winning). Let $a \in \mathsf{Act}_c \cup \{\lambda\}$. If there is a state (l', v') which is a sink state for a, we add a transition (S, a, Bad) in $\tilde{G}$.
- for each $o_i \in \mathcal{O}$ with $o_i \neq \mathsf{Obs}(S)$, if[3] $\mathsf{Next}_a(S) \cap o_i \neq \emptyset$ we add a transition (S, a, S_i) with $S_i = \mathsf{Next}_a(S) \cap o_i$, in $\tilde{G}$.

Given a state S of $\tilde{G}$, we let $\mathsf{Enabled}(S)$ be the set of actions a s.t. (S, a, S') for some S'. A state S of $\tilde{G}$ is winning if its observation is $\mathsf{Obs}(K)$. We let $\tilde{K}$ be the set of winning states of $\tilde{G}$. Notice that $\tilde{G}$ contains only controllable actions. Still $\tilde{G}$ is non-deterministic and thus it can be considered to be a two-player game: from S, the controller chooses an action and the environment chooses the next state among the successors of S by a. This construction of $\tilde{G}$ has the following property:

Theorem 3 ([14]). *The controller has a observation-based stuttering invariant strategy in (G, K) iff there is a winning strategy in $(\tilde{G}, \tilde{K})$.*

We can prove that $(\tilde{G}, \tilde{K})$ is finite and furthermore solving (G, K) amounts to solving a finite non-deterministic game. As we have an efficient algorithm, OTFUR (Fig. 3) to solve this type of games we can solve (G, K). To obtain an efficient algorithm written for (G, K) we simply have to use the Next operator

[3] We let $\mathsf{Next}_a(S) = \cup_{s \in S} \mathsf{Next}_a(s)$.

to compute the transition relation "as needed" in OFTUR. Moreover, in case a state $(l, v) \in S$ is a sink state for a and $\mathsf{Obs}(l, v) = \mathsf{Obs}(S) \neq \mathsf{Obs}(K)$, there is a transition (S, a, Bad) in $\tilde{G}$. We assume that $\mathsf{Obs}(\mathsf{Bad}) = \mathsf{Bad}$ and this observation in not winning so that if the controller plays a from S, then he looses which is consistent as there is a infinite run in G which does not encounter any state with the goal observation. In the version of OFTUR for partially observable timed games it is even more important to propagate backwards the information on losing states like Bad.

The version of the efficient algorithm OTFUR (Fig. 3) for discrete game can be adapted to deal with partially observable games: we obtain a new algorithm OTFPOR given in Fig. 7. In this version, $\mathsf{Sink}_a(S) \neq \emptyset$ stands for "there exists some $(l, v) \in S$ s.t. (l, v) is a sink state for a".

Lines 13 to 22 consist in determining the status of the new symbolic state W' and push it into *ToExplore* or *ToBackPropagate*. Lines 24 and 28 are rather expensive as we have to compute all the symbolic successors of W to determine its new status.

5 Conclusion & Future Work

In [13] we have proposed an efficient for the analysis of reachability timed games. It has been implemented in the tool Uppaal-TiGA [8]. Recently in [14] we have proposed a version of this algorithm for partially observable timed games which is currently being implemented.

There are various directions in which this work can be extended:

- for finite state games, we can generalize our results for reachability games to safety games or more general games with Büchi objectives for instance. This can lead to efficient on-the-fly algorithm for finite games. Also we would like to study particular versions of Büchi objectives, *e.g.* with one repeated location as we may obtain more efficient algorithms for this case;
- for timed games, we can write a dual algorithm for safety objectives. Even in this case this is not always satisfactory as the controller could win with a so-called *zeno* strategy *i.e.* a strategy with which he keeps the game in a good state by playing infinitely many discrete controllable actions in a finite amount of time [15]. It is thus of great importance to ensure that the controller can win in a fair way. This can be encoded by a control objective which is a strengthened by a Büchi objective: we add a particular clock which is reset when it hits the value 1, and the Büchi objective is to hit 1 infinitely often which ensures time divergence. This Büchi objective can be encoded by a single location being forced infinitely often. If we can design an efficient algorithm for a single repeated state in the finite state case, we may be able to obtain efficient algorithms for synthesizing non-zeno controllers for safety timed systems.

These new efficient algorithms are going to be implemented in the tool Uppaal-TiGA and it is expected that new useful practical results will be obtained for real case-studies.

1: **Initialization**
2: $Passed \leftarrow \{\{s_0\}\}$ // *with* $s_0 = (l_0, \mathbf{0})$;
3: $ToExplore \leftarrow \{(\{s_0\}, \alpha, W') \mid \alpha \in \mathsf{Act}_c \cup \{\lambda\},\ o \in \mathcal{O}, o \neq \mathsf{Obs}(s_0),\ W' = \mathsf{Next}_\alpha(\{s_0\}) \cap o \wedge W' \neq \emptyset\}$;
4: $ToBackPropagate \leftarrow \emptyset$;
5: $Win[\{s_0\}] \leftarrow (\{s_0\} \in \tilde{K}\ ?\ 1 : 0)$;
6: $Losing[\{s_0\}] \leftarrow (\{s_0\} \notin \tilde{K} \wedge (ToExplore = \emptyset \vee \forall \alpha \in \mathsf{Act}_c \cup \{\lambda\}, \mathsf{Sink}_\alpha(s_0) \neq \emptyset)\ ?\ 1 : 0)$;
7: $Depend[\{s_0\}] \leftarrow \emptyset$;
8: **Main**
9: **while** $((ToExplore \cup ToBackPropagate \neq \emptyset)) \wedge Win[\{s_0\}] \neq 1 \wedge Losing[\{s_0\}] \neq 1))$ **do**
10: // *pick a transition from ToExplore or ToBackPropagate*
11: $e = (W, \alpha, W') \leftarrow pop(ToExplore)$ or $pop(ToBackPropagate)$;
12: **if** $W' \notin Passed$ **then**
13: $Passed \leftarrow Passed \cup \{W'\}$;
14: $Depend[W'] \leftarrow \{(W, \alpha, W')\}$;
15: $Win[W'] \leftarrow (W' \in \tilde{K}\ ?\ 1 : 0)$;
16: $Losing[W'] \leftarrow (W' \notin \tilde{K} \wedge \mathsf{Sink}_\alpha(W') \neq \emptyset\ ?\ 1 : 0)$;
17: **if** $(Losing[W'] \neq 1)$ **then**
18: $NewTrans \leftarrow \{(W', \alpha, W'') \mid \alpha \in \Sigma,\ o \in \mathcal{O},\ W' = \mathsf{Next}_\alpha(W) \cap o \wedge W' \neq \emptyset\}$;
19: **if** $NewTrans = \emptyset \wedge Win[W'] = 0$ **then** $Losing[W'] \leftarrow 1$;
20: $ToExplore \leftarrow ToExplore \cup NewTrans$;
21: **if** $(Win[W'] \vee Losing[W'])$ **then**
22: $ToBackPropagate \leftarrow ToBackPropagate \cup \{e\}$;
23: **else**
24: $Win^* \leftarrow \bigvee_{c \in \mathsf{Enabled}(W)} \bigwedge_{W \xrightarrow{c} W''} Win[W'']$;
25: **if** Win^* **then**
26: $ToBackPropagate \leftarrow ToBackPropagate \cup Depend[W]$;
27: $Win[W] \leftarrow 1$;
28: $Losing^* \leftarrow \bigwedge_{c \in \mathsf{Enabled}(W)} \bigvee_{W \xrightarrow{c} W''} Losing[W'']$;
29: **if** $Losing^*$ **then**
30: $ToBackPropagate \leftarrow ToBackPropagate \cup Depend[W]$;
31: $Losing[W] \leftarrow 1$;
32: **if** $(Win[W'] = 0 \wedge Losing[W'] = 0)$ **then**
33: $Depend[W'] \leftarrow Depend[W'] \cup \{e\}$;
34: **endif**
35: **endwhile**

Fig. 7. OTFPOR: **O**n-**T**he-**F**ly Algorithm for **P**artially **O**bservable **R**eachability

Acknowledgements

The author wishes to thank Didier Lime and Jean-François Raskin for their careful reading and useful comments on preliminary versions of this paper.

References

1. Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. *Theor. Comput. Sci.*, 354(2):272–300, 2006.
2. K. Altisen, G. Gossler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *IEEE Real-Time Systems Symposium*, pages 154–163, 1999.
3. K. Altisen and S. Tripakis. Tools for controller synthesis of timed systems. In *Proc. 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, 2002. Proc. published as Technical Report 2002-025, Uppsala University, Sweden.
4. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science (TCS)*, 126(2):183–235, 1994.
5. R. Alur, S. La Torre, and G. J. Pappas. Optimal paths in weighted timed automata. In *Proc. 4th International Workshop on Hybrid Systems: Computation and Control (HSCC'01)*, volume 2034 of *LNCS*, pages 49–62. Springer, 2001.
6. E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Proc. 2nd International Workshop on Hybrid Systems: Computation and Control (HSCC'99)*, volume 1569 of *LNCS*, pages 19–30. Springer, 1999.
7. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier Science, 1998.
8. G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *Proceedings of 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 121–125, Berlin, Germany, 2007. Springer.
9. P. Bouyer, F. Cassez, E. Fleury, and K. Larsen. Optimal Strategies in Priced Timed Game Automata. BRICS Reports Series RS-04-0, BRICS, Denmark, Aalborg, Denmark, Feb. 2004. ISSN 0909-0878.
10. P. Bouyer and F. Chevalier. On the control of timed and hybrid systems. *EATCS Bulletin*, 89:79–96, June 2006.
11. P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In W. A. Hunt, Jr and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 180–192, Boulder, Colorado, USA, July 2003. Springer.
12. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: a Model-Checking Tool for Real-Time Systems. In *Proc. 10^{th} Conf. on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.
13. F. Cassez, A. David, E. Fleury, K. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In M. Abadi and L. de Alfaro, editors, *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of *LNCS*, pages 66–80, San Francisco, CA, USA, Aug. 2005. Springer.

14. F. Cassez, A. David, K. Larsen, D. Lime, and J.-F. Raskin. Timed Control with Observation Based and Stuttering Invariant Strategies. In *Proc. of the 5th Int. Symp. on Automated Technology for Verif ication and Analysis (ATVA'2007)*, *LNCS*, Tokyo, Oct. 2007. Springer-Verlag. Forthcoming.
15. F. Cassez, T. A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *Proc. 5th International Workshop on Hybrid Systems: Computation and Control (HSCC'02)*, volume 2289 of *LNCS*, pages 134–148. Springer, 2002.
16. K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for omega-regular games with imperfect information' . In *CSL*, pages 287–302, 2006.
17. L. de Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In *Proc. 12th International Conference on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 536–550. Springer, 2001.
18. S. La Torre, S. Mukhopadhyay, and A. Murano. Optimal-reachability and control for acyclic weighted timed automata. In *Proc. 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002)*, volume 223 of *IFIP Conference Proceedings*, pages 485–497. Kluwer, 2002.
19. F. Laroussinie and K. G. Larsen. CMC: A tool for compositional model-checking of real-time systems. In S. Budkowski, A. R. Cavalli, and E. Najm, editors, *Proceedings of IFIP TC6 WG6.1 Joint Int. Conf. FORTE'XI and PSTV'XVIII*, volume 135 of *IFIP Conference Proceedings*, pages 439–456, Paris, France, Nov. 1998. Kluwer Academic Publishers.
20. F. Laroussinie, K. G. Larsen, and C. Weise. From timed automata to logic – and back. In *Proc. 20th International Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, volume 969 of *LNCS*, pages 529–539. Springer, 1995.
21. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal of Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.
22. X. Liu and S. Smolka. Simple Linear-Time Algorithm for Minimal Fixed Points. In *Proc. 26^{th} Conf. on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.
23. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900 of *LNCS*, pages 229–242. Springer, 1995.
24. P. Ramadge and W. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, 1989.
25. J. Rasmussen, K. G. Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 220–235. Springer, 2004.
26. W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900 of *LNCS*, pages 1–13. Springer, 1995. Invited talk.
27. S. Tripakis and K. Altisen. On-the-Fly Controller Synthesis for Discrete and Timed Systems. In *Proc. of World Congress on Formal Methods (FM'99)*, volume 1708 of *LNCS*, pages 233–252. Springer, 1999.
28. H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *Proc. 36th IEEE Conference on Decision and Control*, pages 4607–4612. IEEE Computer Society Press, 1997.
29. M. D. Wulf, L. Doyen, and J.-F. Raskin. A lattice theory for solving games of imperfect information. In *HSCC*, pages 153–168, 2006.